

FIG. 1

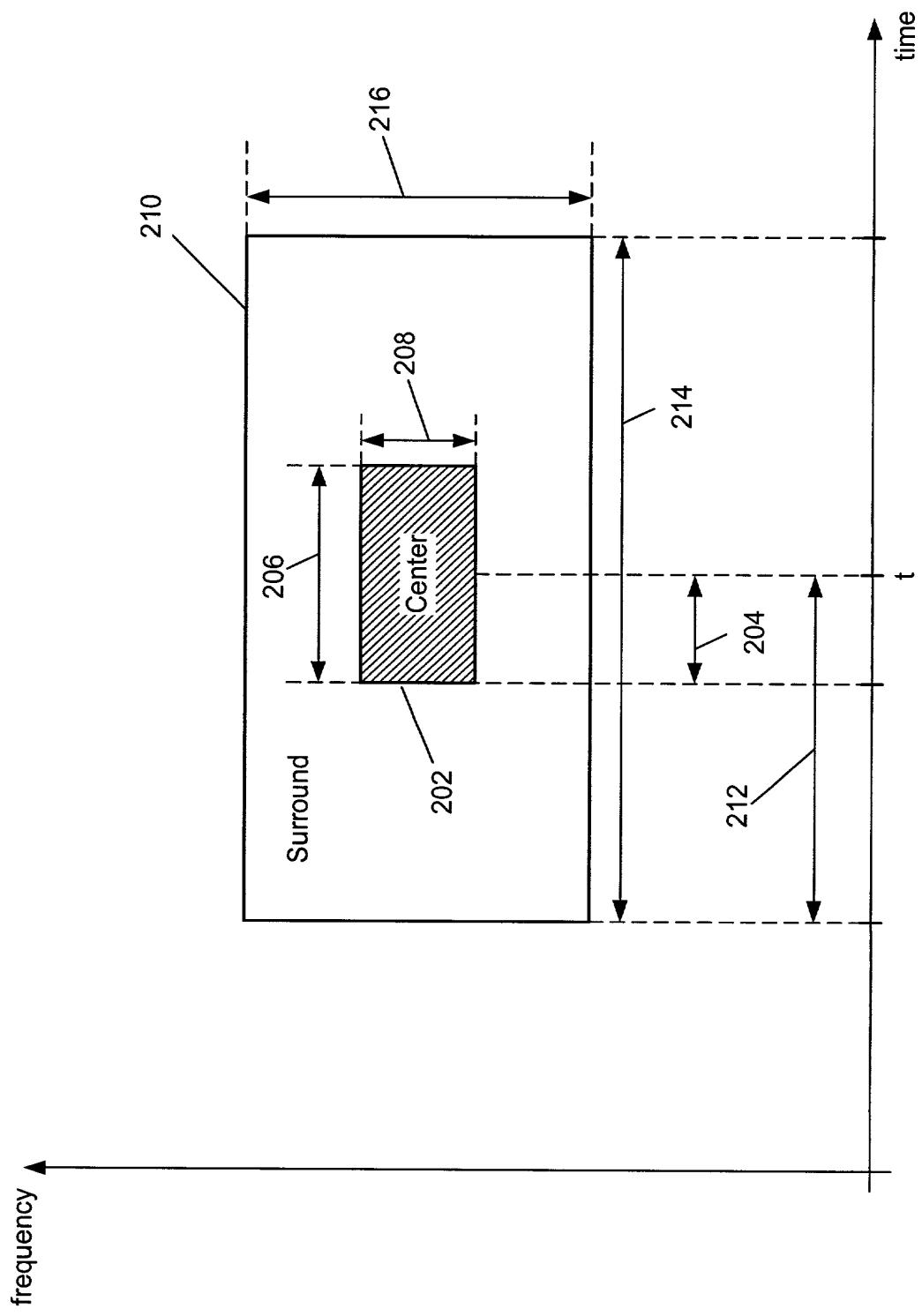


FIG. 2

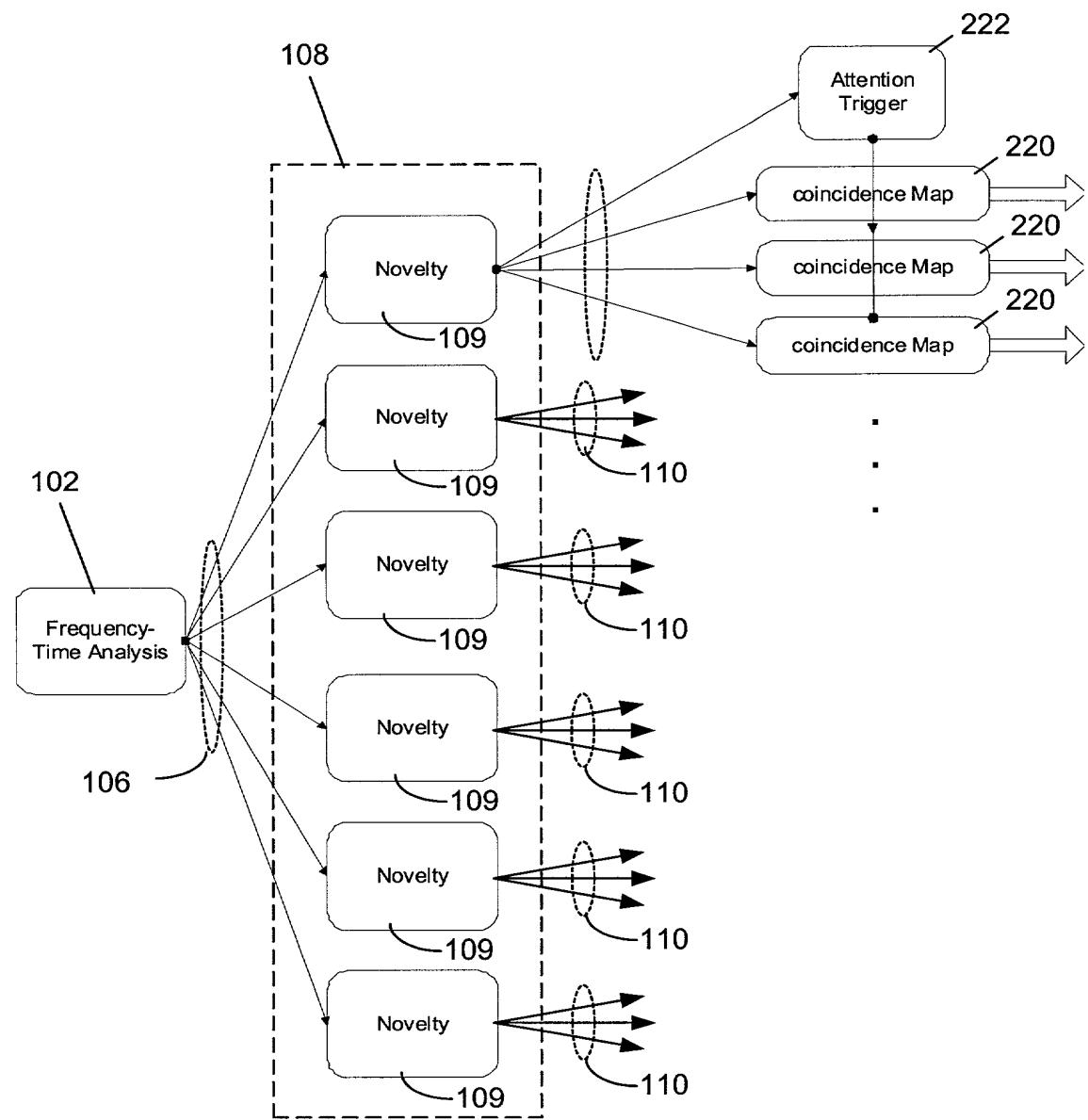


FIG. 3

FRICATIVES

Novelty Processing

Channel	Center Start	Center Length	Center Width	Alpha	Surround Start	Surround Length	Surround Width
0	10	6	1	0.6	0	all	1
1	4	3	8	0.7	6	8	3
2	0	4	2	0.7	1	8	10
3	1	2	2	1.0	0	all	1
4	10	4	4	0.9	0	all	1
5	0	1	10	1.1	4	2	14

FIG. 4

Coincidence Processing. The output size is 576

Function	Attention Trigger	Delta	Start	Stop	Width	Novelty Channel
eCrossColumn	ePlus	-1	1	12	7	0
eCrossColumn	eMinus	1	0	14	13	2
eCrossColumn	eDeltaMinus	-6	6	24	6	2
eCrossColumn	eDeltaPlus	1	12	23	3	2
eCrossColumn	eDeltaMinus	0	3	24	2	3
eCrossColumn	eDeltaPlusM2	-5	5	24	1	3
eCrossColumn	eMinus	-5	22	24	8	4
eCrossColumn	ePlus	0	8	13	1	4
eCrossColumn	eDeltaPlusP2	-7	7	24	6	5
eCrossColumn	eDeltaPlus	3	0	21	16	5
eCrossColumn	ePlus	-5	8	24	16	5
selfAddLocalFreq	eDeltaMinus		7	24	6	5
selfAddLocalFreq			14	24	10	5
selfAddLocalFreq	eMinus		1	24	5	5
selfAddLocalFreq	eDeltaPlus		1	24	8	2
selfAddLocalFreq	eDeltaMinus		0	22	13	1
selfAddLocalFreq	eMinus		0	7	6	1
selfAddLocalFreq	eDeltaMinus		3	24	11	0
crossAddLocalFreq	eDeltaPlus	2	0	22	9	1
crossAddLocalFreq	eDeltaMinus	3	0	21	15	1
crossAddLocalFreq	eDeltaPlusP2	2	0	22	6	2
crossAddLocalFreq		1	13	23	11	2
crossAddLocalFreq		-3	3	24	5	3
crossAddLocalFreq		-1	1	24	3	3
crossAddLocalFreq	ePlus	-4	4	24	12	5
crossAddLocalFreq	ePlus	3	0	21	11	5
crossAddLocalFreq	eMinus	-2	2	24	11	5

FIG. 5

VOWELS

Novelty processing.

Channel	Center Start	Center Length	Center Width	Alpha	Surround Start	Surround Length	Surround Width
0	6	4	4	0.6	0	8	4
1	0	2	1	1.0	0	all	1
2	4	6	6	0.9	0	all	1
3	8	6	3	0.8	8	16	20
4	0	3	6	1.2	2	4	14
5	4	1	1	0.9	2	4	12

FIG. 6

Coincidence Processing. The output size is 696.

Function	Attention Trigger	Delta	Start	Stop	Width	Novelty Channel
EcrossColumn	ePlus	-7	7	22	9	1
EcrossColumn	eDeltaPlusM2	-2	2	24	5	1
ECrossColumn	ePlus	2	0	21	3	1
ECrossColumn	eMinus	-7	17	21	4	2
ECrossColumn	eDeltaMinus	-4	4	24	13	2
ECrossColumn	eDeltaPlus	-7	7	24	6	3
ECrossColumn		-7	7	12	6	3
ECrossColumn	eMinus	-6	6	24	4	3
ECrossColumn	eDeltaMinus	-2	2	24	10	4
SelfAddLocalFreq	eDeltaPlusP2		5	23	16	4
SelfAddLocalFreq	ePlus		2	24	3	5
SelfAddLocalFreq	eDeltaMinus		6	24	16	5
SelfAddLocalFreq	eDeltaMinus		0	21	16	0
SelfAddLocalFreq			3	24	6	1
SelfAddLocalFreq	ePlus		0	24	9	1
CrossAddLocalFreq		-4	4	24	5	1
CrossAddLocalFreq	eDeltaPlus	-4	4	24	7	1
CrossAddLocalFreq	eDeltaPlus	-3	3	23	5	2
CrossAddLocalFreq	ePlus	2	0	22	7	2
CrossAddLocalFreq	ePlus	-2	2	24	5	3
CrossAddLocalFreq	eMinus	-3	3	24	13	3
CrossAddLocalFreq	eDeltaPlusP2	1	0	23	8	3
CrossAddLocalFreq	eMinus	1	0	23	5	4
CrossAddLocalFreq	eDeltaPlus	-2	2	24	6	4
CrossAddLocalFreq	ePlus	-2	2	24	4	5
CrossAddLocalFreq	eMinus	-3	3	24	9	5

FIG. 7

NONFRICATIVES

Novelty Processing.

Channel	Center Start	Center Length	Center Width	Alpha	Surround Start	Surround Length	Surround Width
0	4	4	1	1.0	3	2	3
1	4	4	8	0.6	0	All	1
2	0	2	1	1.1	0	3	10
3	6	6	4	0.7	0	All	1
4	1	2	2	0.6	0	All	1
5	1	4	6	1.2	10	20	8

FIG. 8

Coincidence Processing. The output size is 697.

Function	Attention Trigger	Delta	Start	Stop	Width	Novelty Channel
eCrossColumn	eDeltaPlus	-7	7	16	10	0
eCrossColumn	eMinus	0	0	23	10	0
eCrossColumn		-2	2	24	4	0
eCrossColumn	ePlus	-7	7	17	6	1
eCrossColumn	eDeltaPlus	-1	14	24	10	1
eCrossColumn	eDeltaPlus	1	0	23	2	2
eCrossColumn	eDeltaMinus	0	0	24	4	2
eCrossColumn	eDeltaPlus	-1	1	24	13	2
eCrossColumn	ePlus	2	0	18	10	4
eCrossColumn	eMinus	-5	10	24	5	5
selfAddLocalFreq	ePlus		4	18	17	0
selfAddLocalFreq	eDeltaMinus		0	24	5	0
selfAddLocalFreq	eDeltaPlusM2		5	23	6	1
selfAddLocalFreq			1	24	4	2
crossAddLocalFreq	eMinus	3	0	21	5	0
crossAddLocalFreq	ePlus	-2	2	24	12	0
crossAddLocalFreq		-4	4	24	6	2
crossAddLocalFreq		1	0	23	5	2
crossAddLocalFreq		-2	2	24	5	3
crossAddLocalFreq	eDeltaPlus	1	0	23	6	4
crossAddLocalFreq		-4	4	24	9	4
crossAddLocalFreq		-7	7	24	8	4
crossAddLocalFreq	eDeltaPlus	-2	2	24	3	4
crossAddLocalFreq	eDeltaPlusP2	-3	3	24	10	4
crossAddLocalFreq		-6	6	24	13	5
crossAddLocalFreq	eDeltaPlus	2	9	22	13	5

FIG. 9

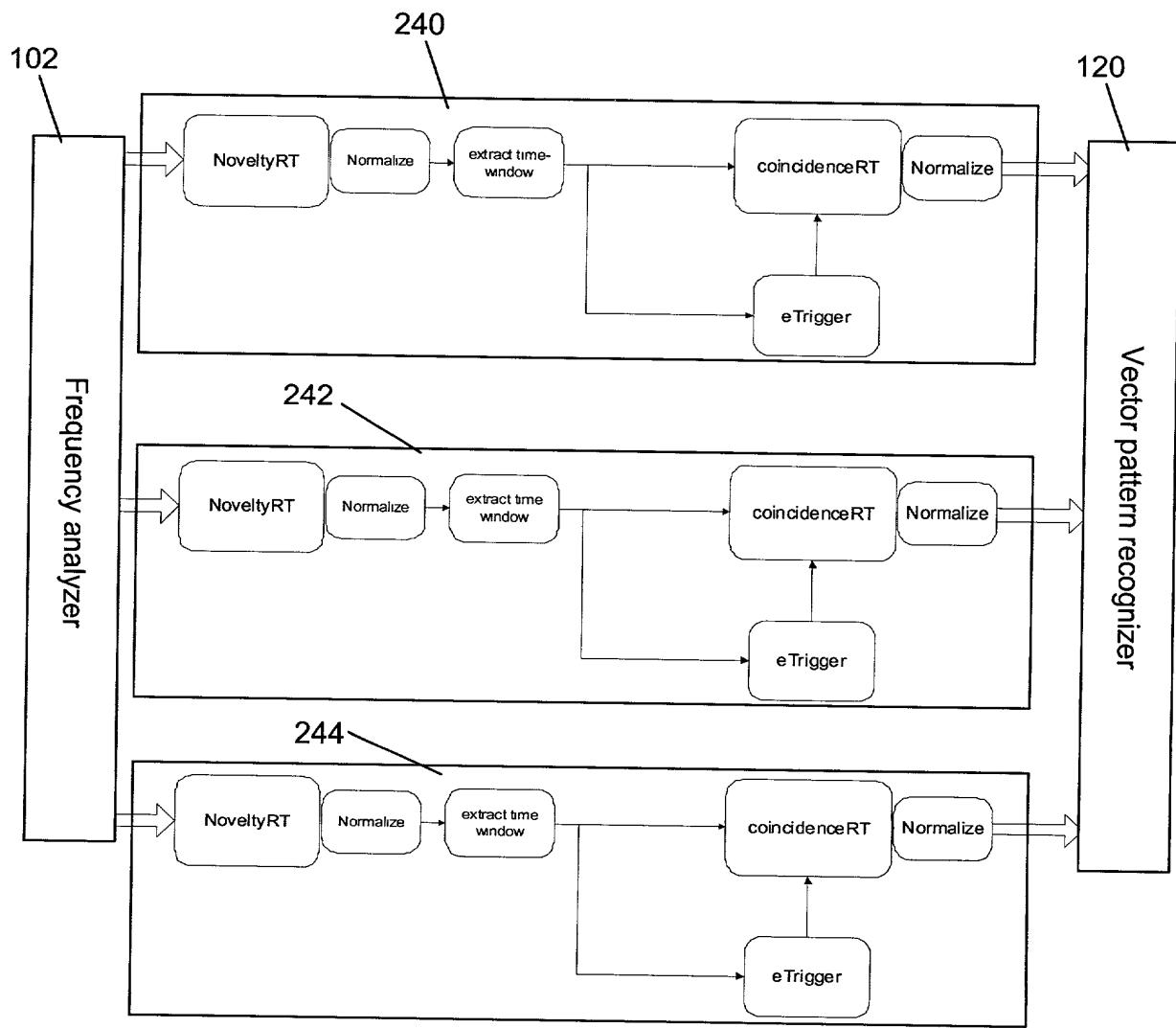


FIG. 10

FIG. 11A

```

v;, f: ---> concatenate --->
, nf: ---> concatenate --->
      ---> mean_square_norm ---> ms:   with  yes yes
ms:   ---> mv_multiply --->      with  k.GA
      ---> map --->      with  npll.GA
      ---> nBest --->best:   with  7

eng:   ---> extract --->      with  3 24 23
      ---> record_stats ---> eLong: with  sum normalize

eLong:   ---> log2 --->      with  1.0
, best:   ---> scale ---> ebest: with  long 1000.0 200.0
spf:   ---> delay --->
      ---> extract --->
      ---> scale ---> sps:
      ---> sps:,ebest: ---> concatenate --->
      ---> putoutAS
      stop

```

FIG. 11B

List of Processes

chunkify	- aggregates the sound data stream into overlapping segments
coincidenceRT	- see main text
compress	- aggregates each data vector into the given bins by averaging over bin
concatenate	- concatenates each pair of incoming vectors into one output vector
cvpower	- computes power spectrum from complex DFT spectrum
delay	- delays a data stream
eTrigger	- see main text
echocancel	- performs echocancelling using voice in and voice out streams
extract	- extracts a time window of specified width and decimation
hanning	- standard hanning filter
log2	- log base2
map	- applies a nonlinear, pointwise transform as specified by control file
mean_square_norm	- normalizes each vector by mean and standard deviation
mu_lawRT	- inverse mu-law
mv_multiply	- matrix-vector multiply
nBest	- zeros out all but the N highest elements of a vector
normalize	- subtracts constant mean vector, and divides by a constant scale vector
noveltyRT	- see main text
putinmaxflo	- provides input data stream from system
putoutAS	- accepts output phonetic stream to pass on to rest of system
record_stats	- computes mean and sigma for each vector
remove_mean	- subtracts the vector mean from each vector
procrustes	- selects a contiguous central portion of a vector
zeroOnLow	- zeros a vector according to empirical low energy condition

Parameter Files

Four4_57.tab	- Fourier coefficient matrix
k.GA	- matrix of phonetic vector-coefficients
mu1fricatives.log40	- constant mean vector
mu1nonfricatives.log40	- constant mean vector
mu1vowels.log40	- constant mean vector
mufricatives.p4	- constant mean vector
munonfricatives.p4	- constant mean vector
muvowels.p4	- constant mean vector
npllr.GA	- specifies nonlinear transform to create log-likelihood ratio output
si1fricatives.log40	- constant scale vector
si1nonfricatives.log40	- constant scale vector
si1vowels.log40	- constant scale vector
sifricatives.p4	- constant scale vector
sinonfricatives.p4	- constant scale vector
sivowels.p4	- constant scale vector

FIG. 12

The ScaleMean object as referenced by normalizeRT.

```
#include "StdAfx.h"
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <math.h>

#define MAX_MEM 100

class ScaleMean
{
    Zbuffer *buf0;
    int start;
    int length;
    int width;
    int numberInBuf0;
    float* fInTime[ MAX_MEM ];

public:
    ScaleMean();
    float get( int i );
    void init( Zipper* aZipper, int aStart, int aLength, int aWidth );
    void update( Zipper* z1 );
    float sides( Zipper* z1, int i );
};

ScaleMean::ScaleMean()
{
    buf0 = NULL;
}

void ScaleMean::init( Zipper* aZipper, int aStart, int aLength, int aWidth )
{
    start = aStart;
    length = aLength;
    width = aWidth;
    numberInBuf0 = 0;
    if( buf0 == NULL )
        buf0 = new Zbuffer( "buf0", aZipper->getType(), aZipper->getSize(),
start+length );
    else
        buf0->zero();
}
```

FIG. 13A

```

float ScaleMean::sides( Zipper* z1, int i )
{
    float sidesSum = (float) 0.0;
    float sidesMean;
    float* pf = (float*) z1->getData();

    // Energy has no sides
    if( i == 0 )
        return pf[0];

    int n = 0;
    int size = (int) z1->getSize();
    // Add in sides where possible.
    int f = max( 1, i - width );
    int fEnd = min( size-1, i + width );
    while ( f <= fEnd ) {
        sidesSum += pf[f++];
        n++;
    }

    sidesMean = (float) quo( sidesSum, ( float ) n );
    return sidesMean;
}

void ScaleMean::update( Zipper* z1 )
{
    buf0->update();

    Zipper* z0 = buf0->get( buf0->getLength()-1 );
    float *pf = (float*) z0->getData();
    for ( long i = 0; i < (long) z1->getSize(); i++ ) {
        pf[i] = sides( z1, i );
    }
    if (numberInBuf0 < (int) buf0->getLength() ) numberInBuf0++;
    int len = buf0->getLength();
    for (int j = 0; j < len; j++ )
    {
        // Find the time in the past.
        Zipper *pz0 = buf0->get(len - 1 - j );
        fInTime[ j ] = ( float* ) pz0->getData();
    }
}

```

FIG. 13B

```
float ScaleMean::get( int i )
{
    float scaleSum = (float) 0.0;
    float scaleMean;

    // Just average over the ones we actually have.
    float n = 0.0;
    for ( int j = start; j < numberInBuf0; j++ ) {
        float* fIn = fInTime[j];
        scaleSum += fIn[i];
        n++;
    }
    scaleMean = (float) quo( scaleSum, n );
    return scaleMean;
}
```

FIG. 13C

The NoveltyRT process. See Attachment 2 for the definition of the ScaleMean object.

Usage -

data:,speaking: ---> NoveltyRT ---> with whichLabelset

```
/*
#include "StdAfx.h"
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <math.h>

#include "speedObject.h"
#include "scalemean.h"

#define MAX_SCALES 10

class NoveltyRT : public SpeedProcess
{
public:
    NoveltyRT();
    ~NoveltyRT();

    int begin();
    int beginFile(Zipper* z1, Zipper* z2 );
    Zipper* processZipper( Zipper* z1, Zipper* z2 );
    Zipper* finalFileZipper();

    void accumMeanAllTime();
    float getScaleMeanAllTime( int i );

    void universal( int size, ScaleMean* center, ScaleMean* surround, double alpha );

private:
    Zbuffer *buf0;

    int numberScales;
    int outSize;
    int whichFlavor;
    BOOL bSpeaking;

    ScaleMean* cScales[ MAX_SCALES ];
    ScaleMean* sScales[ MAX_SCALES ];
    double alpha[ MAX_SCALES ];

    Zbuffer *allTimeSum;
```

FIG. 14A

```

Zipper* pZallTimeSum;
float allTimeCount;

int iout;
Zipper* theZout;
void put( double x ) { theZout->put( iout++, x ); };
};

NoveltyRT::NoveltyRT() : SpeedProcess( "NoveltyRT" )
{
    allTimeSum = NULL;
    buf0 = NULL;
}

NoveltyRT::~NoveltyRT()
{
    delete allTimeSum;
}

/* NoveltyRT - called from initNet() in netnode.cpp
 * whenever a node is created for this process.
 */
SpeedObject* noveltyRT()
{
    return (new NoveltyRT);
}

int NoveltyRT::begin ()
{
    whichFlavor = (int) parameters->askWords( "fricatives nonfricatives vowels" );
    allTimeSum = NULL;
    buf0 = NULL;
    for ( int i=0; i < MAX_SCALES; i++ ) {
        cScales[i] = NULL;
        sScales[i] = NULL;
    }
    return TRUE;
}

Zipper* NoveltyRT::finalFileZipper ()
{
    cout << "Spectrum for Transaction:" << endl;
    if ( allTimeCount > 0 )
        for ( int i = 0; i < pZallTimeSum->getSize(); i++ )
            cout << pZallTimeSum->get(i)/allTimeCount << " ";
}

```

FIG. 14B

```

cout << endl;

    dataState = BUF_EOT;
    return NULL;
}

int NoveltyRT::beginFile(Zipper* z1, Zipper* z2)
{
    assert( z1->getType() == ELIZA_FLOAT );
    for ( int i=0; i < MAX_SCALES; i++ ) {
        delete cScales[i];
        delete sScales[i];
        cScales[i] = NULL;
        sScales[i] = NULL;
    }

    // FRICATIVES
    if ( whichFlavor == 0 ) {
        numberScales = 6;
        cScales[0] = new ScaleMean();
        cScales[0]->init( z1, 10, 6, 1 );
        sScales[0] = NULL;
        alpha[0] = 0.6;

        cScales[1] = new ScaleMean();
        cScales[1]->init( z1, 4, 3, 8 );
        sScales[1] = new ScaleMean();
        sScales[1]->init( z1, 6, 8, 3 );
        alpha[1] = 0.7;

        cScales[2] = new ScaleMean();
        cScales[2]->init( z1, 0, 4, 2 );
        sScales[2] = new ScaleMean();
        sScales[2]->init( z1, 1, 8, 10 );
        alpha[2] = 0.7;

        cScales[3] = new ScaleMean();
        cScales[3]->init( z1, 1, 2, 2 );
        sScales[3] = NULL;
        alpha[3] = 1.0;

        cScales[4] = new ScaleMean();
        cScales[4]->init( z1, 10, 4, 4 );
        sScales[4] = NULL;
        alpha[4] = 0.9;
    }
}

```

FIG. 14C

```

cScales[5] = new ScaleMean();
cScales[5]->init( z1, 0, 1, 10 );
sScales[5] = new ScaleMean();
sScales[5]->init( z1, 4, 2, 14 );
alpha[5] = 1.1;
}

// VOWELS
else if ( whichFlavor == 2 ) {
    numberOfScales = 6;

    cScales[0] = new ScaleMean();
    cScales[0]->init( z1, 6, 4, 4 );
    sScales[0] = new ScaleMean();
    sScales[0]->init( z1, 0, 8, 4 );
    alpha[0] = 0.6;

    cScales[1] = new ScaleMean();
    cScales[1]->init( z1, 0, 2, 1 );
    sScales[1] =NULL;
    alpha[1] = 1.0;

    cScales[2] = new ScaleMean();
    cScales[2]->init( z1, 4, 6, 6 );
    sScales[2] =NULL;
    alpha[2] = 0.9;

    cScales[3] = new ScaleMean();
    cScales[3]->init( z1, 8, 6, 3 );
    sScales[3] = new ScaleMean();
    sScales[3]->init( z1, 8, 16, 20 );
    alpha[3] = 0.8;

    cScales[4] = new ScaleMean();
    cScales[4]->init( z1, 0, 3, 6 );
    sScales[4] = new ScaleMean();
    sScales[4]->init( z1, 2, 4, 14 );
    alpha[4] = 1.2;

    cScales[5] = new ScaleMean();
    cScales[5]->init( z1, 4, 1, 1 );
    sScales[5] = new ScaleMean();
    sScales[5]->init( z1, 2, 4, 12 );
    alpha[5] = 0.9;
}

```

FIG. 14D

```

//NONFRICATIVES
else if ( whichFlavor == 1 ) {
    numberOfScales = 6;

    cScales[0] = new ScaleMean();
    cScales[0]->init( z1, 4, 4, 1 );
    sScales[0] = new ScaleMean();
    sScales[0]->init( z1, 3, 2, 3 );
    alpha[0] = 1.0;

    cScales[1] = new ScaleMean();
    cScales[1]->init( z1, 4, 4, 8 );
    sScales[1] = NULL;
    alpha[1] = 0.6;

    cScales[2] = new ScaleMean();
    cScales[2]->init( z1, 0, 2, 1 );
    sScales[2] = new ScaleMean();
    sScales[2]->init( z1, 0, 3, 10 );
    alpha[2] = 1.1;

    cScales[3] = new ScaleMean();
    cScales[3]->init( z1, 6, 6, 4 );
    sScales[3] = NULL;
    alpha[3] = 0.7;

    cScales[4] = new ScaleMean();
    cScales[4]->init( z1, 1, 2, 2 );
    sScales[4] = NULL;
    alpha[4] = 0.6;

    cScales[5] = new ScaleMean();
    cScales[5]->init( z1, 1, 4, 6 );
    sScales[5] = new ScaleMean();
    sScales[5]->init( z1, 10, 20, 8 );
    alpha[5] = 1.2;
}

allTimeCount = (float) 0.0;
outSize = z1->getSize() * numberOfScales;

if ( buf0 == NULL )
{
    // We need a min of 3 for finding energy in accumMeanAllTime.
    buf0 = new Zbuffer ( "buf0", z1->getType(), z1->getSize(), 3 );
}

```

FIG. 14E

```

        allTimeSum = new Zbuffer( "allTimeSum", z1->getType(), z1->getSize(), 1
);
        pZallTimeSum = allTimeSum->get( 0 );

    }
    else
        allTimeSum->zero();

    return TRUE;
}

/* Returns the mean for all time in this file.
 */
void NoveltyRT::accumMeanAllTime()
{
    // Get average of energy AROUND time 1.
    float energy = (float) 0.0;
    for (int i = 0; i < 3; i++)
        energy += (float) buf0->get(i)->get(0);
    energy = (float) (energy / 3.0);

    /* Check fixed energy threshold. */
    if (energy > 22)
    {
        float *pfSum = (float*) pZallTimeSum->getData();
        allTimeCount++;
        for ( i = 0; i < (int) pZallTimeSum->getSize(); i++ )
        {
            float newVal = (float) buf0->get(1)->get(i);
            pfSum[i] += newVal;
        }
    }

    return;
}

float NoveltyRT::getScaleMeanAllTime( int i )
{
    float scaleSum = (float ) pZallTimeSum->get( i );
    return (float) quo( scaleSum, allTimeCount );
}

Zipper* NoveltyRT::processZipper(Zipper* z1, Zipper* z2)
{
    Zipper* zout = Zipper::createZipper( ELIZA_FLOAT, outSize );
    zout->zero();
}

```

FIG. 14F

```

bSpeaking = FALSE;
if(z2) bSpeaking = int(z2->get(0));
// Store the input data in memory.
buf0->put( z1 );
if(bSpeaking) {

} else
    accumMeanAllTime();

// compute the multiscale NoveltyRT for each input point.
iout = 0;
theZout = zout;

for (int b = 0; b < numberOfScales; b++) {
    if ( cScales[b] != NULL )
        cScales[b]->update( z1 );
    if ( sScales[b] != NULL )
        sScales[b]->update( z1 );
    universal( z1->getSize(), cScales[b], sScales[b], alpha[b] );
}
return zout;
}

void NoveltyRT::universal( int size, ScaleMean* center, ScaleMean* surround, double
alpha )
{
    double scaleMean, scaleMean2, theNovelty;

    if ( surround == NULL ) {
        for (int i = 0; i < size; i++ ) {
            scaleMean = center->get( i );
            scaleMean2 = getScaleMeanAllTime( i );
            theNovelty = scaleMean - alpha * scaleMean2;
            put( theNovelty );
        }
    }
    else {
        for (int i = 0; i < size; i++ ) {
            scaleMean = center->get( i );
            scaleMean2 = surround->get( i );
            theNovelty = scaleMean - alpha * scaleMean2;
            put( theNovelty );
        }
    }
}

```

FIG. 14G

Usage -

---> coincidenceRT ---> with name0 | name1 | etc.

Function -

```
/*
#include "StdAfx.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <math.h>

#include "speedObject.h"
#include "eTrigger.h"

class CoincidenceRT : public SpeedProcess
{
public:
    CoincidenceRT();
    ~CoincidenceRT();

    int begin();
    int beginFile(Zipper* z1, Zipper* z2);
    int final();
    Zipper* processZipper (Zipper*, Zipper* );
    double get( int t, int f, int offset )
    {
        if ( t < 0 )
            return 0.0;
        else
            return pin[ t * inputStride + offset + f ];
    }

public:
    int numberOfRowsScales;
    int numberOfRowsTimes;
    int gateStride;
    int columnSize;
    int localFreqSize;

private:
    int whichProcess;
    BOOL countMode;
    BOOL bSetSize;
```

FIG. 15A

```

int nTotal;
    int inputStride;
    int deltaWidth;
    int deltaStep;
    float *pin;
    char* pGate;
    float *pout;

    float* getAddr( int t, int f, int offset ) {
        return pin + (t * inputStride + offset + f);
    }

    void put( double x ) {
        if( countMode )
            nTotal++;
        else
            *pout++ = (float) x;
    }

    void doFricatives2();
    void doNonFricatives2();
    void doVowels2();
    void doGA();
    void doNonFricatives();
    void doVowels();
    void doFricatives();
    void dispatch();

    void eCrossColumn( int delta, int tstart, int tstop, int fWidth, int whichScale );
    void eCrossColumn( eGateType eGate, int delta, int tstart, int tstop, int fWidth, int
whichScale );

    void selfAddLocalFreq(      int tstart, int tstop, int localN, int whichScale );
    void selfAddLocalFreq( eGateType eGate, int tstart, int tstop, int localN, int
whichScale );

    void crossAddLocalFreq( int delta, int tstart, int tstop, int fWidth, int whichScale );
    void crossAddLocalFreq( eGateType eGate, int delta, int tstart, int tstop, int
fWidth, int whichScale );
};

int CoincidenceRT::begin()
{
    deltaWidth = localFreqSize = 4;
    deltaStep = 1;
    numberOfScales = (int) parameters->AskUser("Number of scales from novelty",
0.0, 10.0, 10.0 );
}

```

FIG. 15B

```

        numberOfTimes = (int) parameters->AskUser("Number of times in RF", 0.0,
100.0, 100.0 );
        gateStride = numberOfTimes * numberScales;
        localFreqSize = (int) parameters->AskUser("Local Frequency Size", 0.0, 100.0,
100.0 );
        whichProcess = parameters->askWords("GA nonfricatives vowels fricatives");
        bSetSize = TRUE;
        return TRUE;
    }
void CoincidenceRT::dispatch()
{
    switch (whichProcess) {
        case 0:
            doGA();
            break;
        case 1:
            doNonFricatives();
            break;
        case 2:
            doVowels();
            break;
        case 3:
            doFricatives();
            break;
    }
}

int CoincidenceRT::beginFile(Zipper* z1, Zipper* z2)
{
    // Simulate one run in countMode
    pin = (float *) z1->getData();
    pGate = (char *) z2->getData();
    if (bSetSize)
    {
        columnSize = z1->getSize() / (numberOfScales * numberOfTimes );
        inputStride = numberScales * columnSize;

        nTotal = 0;
        countMode = TRUE;
        dispatch();
        countMode = FALSE;
        int nf = columnSize-1;
        int nf1 = nf / localFreqSize;
        bSetSize = FALSE;
        cout << "CoincidenceRT: stride " << inputStride;
        cout << " columnSize " << columnSize;
    }
}

```

FIG. 15C

```

//cout << " nfl is " << nfl << " nTotal = " << nTotal;
cout << endl;
}
return TRUE;
}

Zipper* CoincidenceRT::processZipper (Zipper* z1, Zipper* z2)
{
    pin = (float *) z1->getData();
    pGate = (char *) z2->getData();
    Zipper* zOut = Zipper::createZipper( ELIZA_FLOAT, nTotal );
    zOut->zero();
    pout = (float *) zOut->getData();

    //dist->increment( ehi[1] - ehi[0]);
    dispatch();
    return zOut;
}

void CoincidenceRT::doFricatives()
{
    //output size is 576
    eCrossColumn( ePlus,      -1, 1, 12, 7, 0 );
    eCrossColumn( eMinus,      1, 0, 14, 13, 2 );
    eCrossColumn( eDeltaMinus, -6, 6, 24, 6, 2 );
    eCrossColumn( eDeltaPlus,   1, 12, 23, 3, 2 );
    eCrossColumn( eDeltaMinus,  0, 3, 24, 2, 3 );
    eCrossColumn( eDeltaPlusM2, -5, 5, 24, 1, 3 );
    eCrossColumn( eMinus,      -5, 22, 24, 8, 4 );
    eCrossColumn( ePlus,       0, 8, 13, 1, 4 );
    eCrossColumn( eDeltaPlusP2, -7, 7, 24, 6, 5 );
    eCrossColumn( eDeltaPlus,   3, 0, 21, 16, 5 );
    eCrossColumn( ePlus,       -5, 8, 24, 16, 5 );
    selfAddLocalFreO( eDeltaMinus, 7, 24, 6, 5 );
    selfAddLocalFreq(           14, 24, 10, 5 );
    selfAddLocalFreO( eMinus,    1, 24, 5, 5 );
    selfAddLocalFreO( eDeltaPlus, 1, 24, 8, 2 );
    selfAddLocalFreO( eDeltaMinus, 0, 22, 13, 1 );
    selfAddLocalFreO( eMinus,    0, 7, 6, 1 );
    selfAddLocalFreO( eDeltaMinus, 3, 24, 11, 0 );
    crossAddLocalFreO( eDeltaPlus, 2, 0, 22, 9, 1 );
    crossAddLocalFreO( eDeltaMinus, 3, 0, 21, 15, 1 );
    crossAddLocalFreO( eDeltaPlusP2, 2, 0, 22, 6, 2 );
    crossAddLocalFreO(           1, 13, 23, 11, 2 );
    crossAddLocalFreO(           -3, 3, 24, 5, 3 );
    crossAddLocalFreO(           -1, 1, 24, 3, 3 );
}

```

FIG. 15D

```

crossAddLocalFreO( ePlus,      -4, 4, 24, 12, 5 );
crossAddLocalFreO( ePlus,      3, 0, 21, 11, 5 );
crossAddLocalFreO( eMinus,     -2, 2, 24, 11, 5 );
}

void CoincidenceRT::doNonFricatives()
{
    //output size is 697
    eCrossColumn( eDeltaPlus, -7, 7, 16, 10, 0 );
    eCrossColumn( eMinus,      0, 0, 23, 10, 0 );
    eCrossColumn(          -2, 2, 24, 4, 0 );
    eCrossColumn( ePlus,      -7, 7, 17, 6, 1 );
    eCrossColumn( eDeltaPlus, -1, 14, 24, 10, 1 );
    eCrossColumn( eDeltaPlus,  1, 0, 23, 2, 2 );
    eCrossColumn( eDeltaMinus, 0, 0, 24, 4, 2 );
    eCrossColumn( eDeltaPlus, -1, 1, 24, 13, 2 );
    eCrossColumn( ePlus,      2, 0, 18, 10, 4 );
    eCrossColumn( eMinus,     -5, 10, 24, 5, 5 );
    selfAddLocalFreO( ePlus,     4, 18, 17, 0 );
    selfAddLocalFreO( eDeltaMinus, 0, 24, 5, 0 );
    selfAddLocalFreO( eDeltaPlusM2, 5, 23, 6, 1 );
    selfAddLocalFreq(          1, 24, 4, 2 );
    crossAddLocalFreO( eMinus,    3, 0, 21, 5, 0 );
    crossAddLocalFreq( ePlus,     -2, 2, 24, 12, 0 );
    crossAddLocalFreO(          -4, 4, 24, 6, 2 );
    crossAddLocalFreO(          1, 0, 23, 5, 2 );
    crossAddLocalFreO(          -2, 2, 24, 5, 3 );
    crossAddLocalFreO( eDeltaPlus, 1, 0, 23, 6, 4 );
    crossAddLocalFreO(          -4, 4, 24, 9, 4 );
    crossAddLocalFreO(          -7, 7, 24, 8, 4 );
    crossAddLocalFreO( eDeltaPlus, -2, 2, 24, 3, 4 );
    crossAddLocalFreq( eDeltaPlusP2, -3, 3, 24, 10, 4 );
    crossAddLocalFreq(          -6, 6, 24, 13, 5 );
    crossAddLocalFreq( eDeltaPlus,  2, 9, 22, 13, 5 );
}

void CoincidenceRT::doVowels()
{
    //output size is 696
    eCrossColumn( ePlus,      -7, 7, 22, 9, 1 );
    eCrossColumn( eDeltaPlusM2, -2, 2, 24, 5, 1 );
    eCrossColumn( ePlus,      2, 0, 21, 3, 1 );
    eCrossColumn( eMinus,     -7, 17, 21, 4, 2 );
    eCrossColumn( eDeltaMinus, -4, 4, 24, 13, 2 );
    eCrossColumn( eDeltaPlus,  -7, 7, 24, 6, 3 );
    eCrossColumn(          -7, 7, 12, 6, 3 );
}

```

FIG. 15E

```

eCrossColumn( eMinus, -6, 6, 24, 4, 3 );
eCrossColumn( eDeltaMinus, -2, 2, 24, 10, 4 );
selfAddLocalFreO( eDeltaPlusP2, 5, 23, 16, 4 );
selfAddLocalFreO( ePlus, 2, 24, 3, 5 );
selfAddLocalFreO( eDeltaMinus, 6, 24, 16, 5 );
selfAddLocalFreO( eDeltaMinus, 0, 21, 16, 0 );
selfAddLocalFreq( 3, 24, 6, 1 );
selfAddLocalFreO( ePlus, 0, 24, 9, 1 );
crossAddLocalFreO( -4, 4, 24, 5, 1 );
crossAddLocalFreO( eDeltaPlus, -4, 4, 24, 7, 1 );
crossAddLocalFreO( eDeltaPlus, -3, 3, 23, 5, 2 );
crossAddLocalFreO( ePlus, 2, 0, 22, 7, 2 );
crossAddLocalFreO( ePlus, -2, 2, 24, 5, 3 );
crossAddLocalFreO( eMinus, -3, 3, 24, 13, 3 );
crossAddLocalFreO( eDeltaPlusP2, 1, 0, 23, 8, 3 );
crossAddLocalFreO( eMinus, 1, 0, 23, 5, 4 );
crossAddLocalFreO( eDeltaPlus, -2, 2, 24, 6, 4 );
crossAddLocalFreO( ePlus, -2, 2, 24, 4, 5 );
crossAddLocalFreO( eMinus, -3, 3, 24, 9, 5 );
}

void CoincidenceRT::doGA()
{
    //doVowels();
    //doFricatives();
//    doNonFricatives();
}

void CoincidenceRT::eCrossColumn( int delta, int tstart, int tstop, int fWidth, int
whichScale=0 )
{
    int scaleBase = whichScale * columnSize;
    // Energy by itself
    double sum = 0.0;
    for ( int t = tstart; t < tstop; t++ )
        sum += get( t, 0, scaleBase ) * get( t+delta, 0, scaleBase );
    put( sum );

    for ( int f = 1; f <= columnSize-fWidth; f += fWidth )
    {
        sum = 0.0;
        for ( int t = tstart; t < tstop; t++ )
        {
            float* p1 = getAddr( t, 0, scaleBase );
            float* p2 = getAddr( t+delta, f, scaleBase );
            for ( int i = 0; i < fWidth; i++ )

```

FIG. 15F

```

        sum += *p1 * *p2++;
    }
    put( sum );
}
}

// N = 1 + numberOfFreqs/fWidth
void CoincidenceRT::eCrossColumn( eGateType eGate, int delta, int tstart, int tstop, int
fWidth, int whichScale=0 )
{
    int scaleBase = whichScale * columnSize;
    int outOffset = whichScale * numberoftimes;
    char* eGateA = pGate + gateStride * eGate + outOffset;

    // Energy by itself
    double sum = 0.0;
    int stop = min( numberoftimes-1-delta, tstop );
    int start = max( 0-delta, tstart );
    for ( int t = start; t < stop; t++ ) {
        sum += get( t, 0, scaleBase ) * get( t+delta, 0, scaleBase );
    }
    put( sum );

    for ( int f = 1; f <= columnSize-fWidth; f += fWidth )
    {
        sum = 0.0;
        for ( int t = start; t < stop; t++ )
        {
            if ( eGateA[t] ) {
                float* p2 = getAddr( t+delta, f, scaleBase );
                for ( int i = 0; i < fWidth; i++ ) {
                    sum += *p2++;
                }
            }
        }
        put( sum );
    }
}

void CoincidenceRT::selfAddLocalFreq( int tstart, int tstop, int localN, int whichScale )
{
    int scaleBase = whichScale * columnSize;

    // Do full self product, but amalgamate by localN
    for ( int f1 = 1; f1 < columnSize-localN; f1 += localN )
    {

```

FIG. 15G

```

for ( int f2 = 1; f2 <= f1; f2 += localN )
{
    double sum = 0.0;
    for ( int t = tstart; t < tstop; t++ )
    {
        float* p1 = getAddr( t, f1, scaleBase );
        float* p2 = getAddr( t, f2, scaleBase );
        for ( int i = 0; i < localN; i++ )
            sum += *p1++ * *p2++;
    }
    put( quo( sum, tstop - tstart ) );
}
}

// N = ( numberOfFreqs / localN ) * ( numberOfFreqs / localN - 1 ) / 2
void CoincidenceRT::selfAddLocalFreq( eGateType eGate, int tstart, int tstop, int fWidth,
int whichScale )
{
    int scaleBase = whichScale * columnSize;
    int outOffset = whichScale * numberOfTimes;
    char* eGateA = pGate + gateStride * eGate + outOffset;

    // Do full self product, but amalgamate by fWidth
    for ( int f1 = 1; f1 < columnSize-fWidth; f1 += fWidth ) {
        for ( int f2 = 1; f2 <= f1; f2 += fWidth ) {
            double sum = 0.0;
            for ( int t = tstart; t < tstop; t++ ) {
                if( eGateA[ t ] ) {
                    float* p1 = getAddr( t, f1, scaleBase );
                    float* p2 = getAddr( t, f2, scaleBase );
                    for ( int i = 0; i < fWidth; i++ ) {
                        sum += *p1++ * *p2++;
                    }
                }
            }
            put( sum );
        }
    }
}

// N = ( numberOfFreqs / fWidth ) **2
void CoincidenceRT::crossAddLocalFreq( int delta, int tstart, int tstop, int fWidth, int
whichScale )
{
    int scaleBase = whichScale * columnSize;
}

```

FIG. 15H

```

// Do full cross product, but amalgamate by 2s
for ( int f1 = 1; f1 <= columnSize - fWidth; f1 += fWidth )
{
    for ( int f2 = 1; f2 <= columnSize - fWidth; f2 += fWidth )
    {
        double sum = 0.0;
        for ( int t = tstart; t < tstop; t++ )
        {
            float* p1 = getAddr( t+delta, f1, scaleBase );
            float* p2 = getAddr( t, f2, scaleBase );
            for ( int i = 0; i < fWidth; i++ )
                sum += *p1++ * *p2++;
        }
        put( sum );
    }
}

```

```

void CoincidenceRT::crossAddLocalFreq( eGateType eGate, int delta, int tstart, int tstop,
int fWidth, int whichScale )
{
    int scaleBase = whichScale * columnSize;
    int outOffset = whichScale * numberOfTimes;
    char* eGateA = pGate + gateStride * eGate + outOffset;

    // Do full cross product, but amalgamate by 2s
    for ( int f1 = 1; f1 <= columnSize - fWidth; f1 += fWidth ) {
        for ( int f2 = 1; f2 <= columnSize - fWidth; f2 += fWidth ) {
            double sum = 0.0;
            for ( int t = tstart; t < tstop; t++ ) {
                if ( !eGateA[t] ) continue;

                float* p1 = getAddr( t+delta, f1, scaleBase );
                float* p2 = getAddr( t, f2, scaleBase );
                for ( int i = 0; i < fWidth; i++ )
                    sum += *p1++ * *p2++;
            }
            put( sum );
        }
    }
}

```

FIG. 15I